N72-30531

DESIGN OF AUTOMATA
THEORY OF CUBICAL COMPLEXES WITH APPLICATIONS TO DIAGNOSIS AND ALGORITHMIC DESCRIPTION

by J. Paul Roth

FINAL REPORT

For research sponsored in part by the Jet Propulsion Laboratory, California Institute of Technology with the support, under Contract NAS 7-100, of the National Aeronautics and Space Administration.

JPL CONTRACT NO. 952709

IBM Research Report No. RC 3814

April 11, 1972

International Business Machines Corporation
IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

DESIGN OF AUTOMATA
THEORY OF CUBICAL COMPLEXES WITH APPLICATIONS TO DIAGNOSIS AND ALGORITHMIC DESCRIPTION

by

J. Paul Roth

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

ABSTRACT: This investigation has been concerned with the following problems: (1) methods for development of logic design together with algorithms such that it is possible by means of these algorithms to compute a test for any failure in the logic design, if such a test exists; it is also concerned with developing algorithms and heuristics for the purpose of minimizing the computation for tests. (2) a method of design of logic for ultra LSI (Large Scale Integration) which seems to make LSI technologically feasible (failures may exist on the chip with correct operations still taking place) This method of design, called the Universal Function Schema, allows for instant Engineering Changes EC's as well as for complete functional changes in milliseconds of time. In addition this scheme provides a "universal card" which solves the so-called "stocking problem" as well as reduces the cost of production. (One must pay the penalty of about a 10 times increase in circuit count) (3) It has been discovered that the so-called quantum calculus can be extended to render it possible: 1) to describe the functional behavior of a mechanism component-by-component and 2) to compute tests for failures, in the mechanism, using the Diagnosis algorithm. In view of the large amount of extant electro-mechanical gear, this result may be of basic importance. (4) One of the original motivations for this investigation was the development of an algorithm for the multi-output 2-level minimization problem. A program MIN 360 (its new name) has been written for this algorithm. MIN 360 has options of mode (exact minimum or various approximations), cost function, cost bound, etc., providing flexibility. MIN 360 has been applied to some of the Jet Propulsion Laboratory's problems; its solutions seem already to have been incorporated into some of JPL's hardware.

This work has been sponsored in part by the Jet Propulsion Laboratory, California Institute of Technology with the support, under Contract NAS 7-100, of the National Aeronautics and Space Administration.

RC 3814 (#17297)
April 11, 1972
Computer Sciences
Mathematics

Section 1. Diagnosible Design Form and the D-algorithm

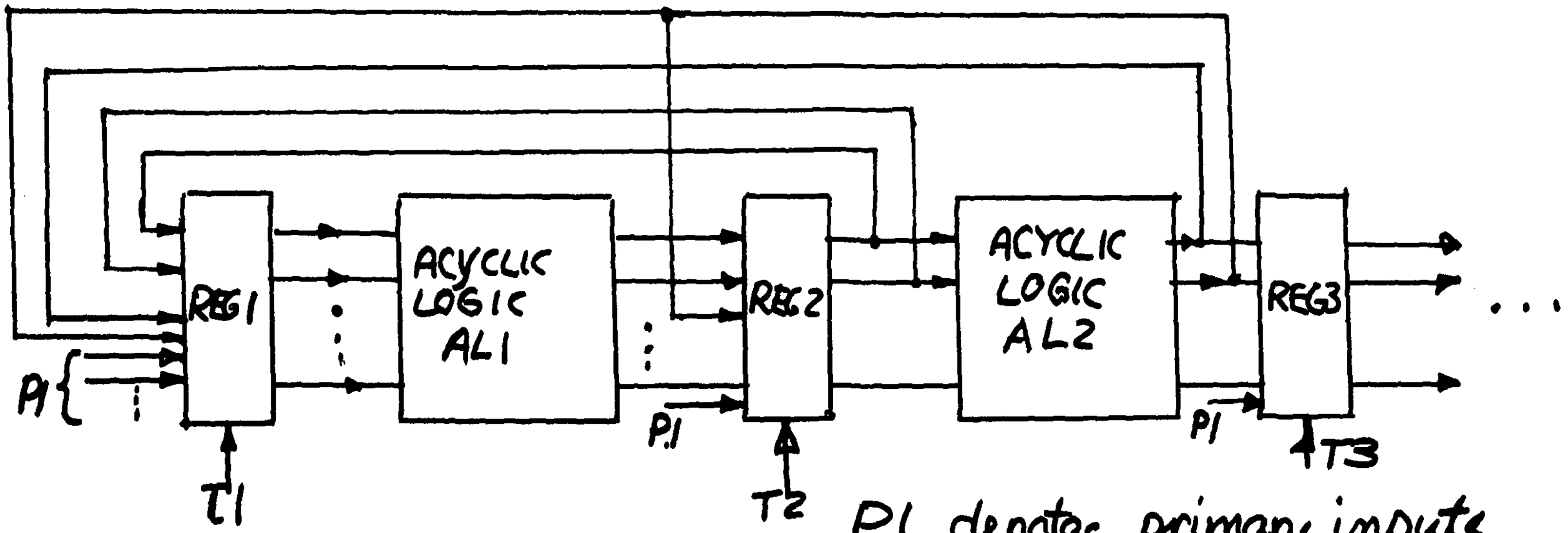
If one designs random asynchronous logic and attempts to develop tests for failures in this logic design, one is doomed to failure if the circuit is at all large. The reason for this is as follows: The D-algorithm (as with all the others of which we know) when confronted with asynchronous cyclic designs attempts to cut certain feedback loops in an adroit manner, so that the cuts appear to be the natural places in which to insert unit delays for all the cuts. One then generates tests for this simplified model of the original circuit, simplified in the sense of its timing behavior. Does the test for failure in the model also represent a test in the original more realistic design? In order to find out, one runs simulators which locate races and hazards. If the test, which in general is a sequence of patterns to the primary inputs of the circuit, has a race then it must be discarded and another attempt to compute a test for another failure. As the size of the circuit increases the frequency with which a "potential test" harbors races increases. The method of design, called Diagnosible Design Form, described in this section, is a procedure for the the design of logic in which it possible to realize any function; furthermore, one can prove that the D-algorithm (extended to these sequential circuits) will always compute a test for a failure if such exists.

System/360 model 40 and the system/370 Model 195 both used almost exclusively diagnosible design form.

Diagnosible Design Form may be described with reference to the figure below. The basic configuration consists of a bank of latches - call them registers - R1 which are gated at time T1. This R1 feeds acyclic logic L1 (hence combinational logic) which is followed by a bank of registers R2 gated at time T2. Feedback is allowed from R2 to R1. R2 in turn is followed by a section of acyclic logic L2, followed by registers R3 gated at time T3 with feedback allowed from R3 to R2 or R1. The difference T2 - T1 is chosen large enough so that all changes in the acyclic logic have taken place from T1 gating R1 before T2 is activated etc. Thus all races hazards are, by design, eliminated.

Our only remaining task is to show that the D-algorithm slightly modified may be easily adapted to computing tests (a deterministic test) for a given testable failure F1. In the first place, it may be noted that the obvious place to cut the feedback loops is from one bank of registers to another, thus there is no complicated choice which must be made.

It was stated that it could be guaranteed that the D-algorithm would compute a "deterministic" test for a given failure if one existed. By deterministic is meant that the tests, in general of sequences of input patterns, do not depend upon the signal in the feedback loops (state variables) at the time of application of the tests. One would have also to compute "D-sequences" and "singular cube sequences" for the latches, a trivial one-shot computation, and make sure that only such tests were used in the generation of a test.



PI denotes primary inputs

Figure 1 Diagnosible Design Form

The D-algorithm for the DDF will be termed DALG5.

Section 2. Heuristics for DALG

The computation for a "covering" ensemble for tests for very large circuits (say the order of 10,000 devices) become significant. The following two heuristics are offered with the purpose of reducing the computation. The first step in the procedure to compute an ensemble of tests to detect any failure of a given category in a logic design is to select a failure out of this given category. The first heuristic is a method for making such a selection. The procedure starts as follows: a failure for a primary input for the circuit is first chosen (we do not have a criterion for selecting among the primary input failures). One then uses the D-algorithm (or some variant or equivalent thereof) to compute a test for this failure. One then uses TESTDETECT to ascertain all failures detected by this test (in general a sequence of input patterns). One would wish to maximize the number of failures detected by a given test for this tends towards minimizing the total number of tests computed and hence minimizing the total computation. Thinking of the D-algorithm in terms of the D-chains developed in generating the tests, for a failure originating in the primary input (a PDCF) this chain drives through the entire logic and it can be shown that any line on this D-chain will be tested in one mode or another by this particular test being generated: for this choice the D-chain tends to be as large as possible since it is generated for a primary input. Next one selects some primary input failure not yet tested if such exists, and repeats the same procedure. If no such primary input failure untested by the tests computed to date exists, one next selects a device fed by primary inputs and goes through the same procedure etc., until an ensemble of tests has been computed which covers all failures of the category agreed upon in advance.

It is estimated that the number of tests required by this mode of computation as opposed to the "counter strategy" of starting with the primary output failures and proceeding backwards to the primary input failures would be about 1 to 2.

The second strategy is a simple one which in complicated control circuits might be extremely helpful. For some of these circuits a sequence of reset input patterns is originally applied in order to render the design into a known and desired state. The strategy followed is thus to first apply the sequence of reset inputs which is known, and then to derive the D-chain emanating from the failing device in the usual way: this imposes a constraint on the development of the D-chain. It is likely that the D-algorithm even with this constraint will compute a test if it exists for the failure. Indeed it may be that it is necessary to apply the reset input sequence before one is able to get a test; the D-algorithm would not know this in advance and thus might take a great deal of time in generating the test if indeed it had enough computation time so to do. It is expected that this short-cut will substantially speed up computation time. The exact algorithm would then be followed by this and used only if the heuristic did not yield a test.

Section 3. Cyclic TESTDETECT

In the computation of an ensemble of tests to detect any failure of a given category in a logic design the procedure is usually as follows: (1) select a failure from a given category of failures and compute a test for it (e.g. by the D-algorithm) (2) use a simulator to determine all failures of the given category which are detected by this test. This procedure is continued until a test ensemble has been obtained which detects all the prescribed failures (or one has reached a certain modicum of completeness or "coverage", say 85%).

The bulk of computation time is in the simulation, perhaps by a ratio of at least 100 to 1.

TESTDETECT was developed as a "1-pass" simulator to determine all failures detected by a given test. It was designed originally for acyclic (combinational) logic. It had not been seen how to the generalize TESTDETECT to the general cyclic circuit case. A study of the Diagnosible Design Form, however, shows that if the logic design is clocked by a clock having at least two phases then it is rather simple to extend TESTDETECT to cover DDF-logic.

In order to describe cyclic TESTDETECT we will first review how it works for acyclic logic. Assume that we have a test consisting of a single wave of patterns applied to the primary inputs of the circuit. First one computes the signal on each line (device) in the logic design. Let us now look at the primary output. If a given output has a "0" assigned to it for this input pattern, then clearly it is tested for stuck-at-1. Likewise if it has a 1, then it is tested for stuck-at-0. Assume that a given primary output is a 2-input OR with inputs 1 and 0 respectively. It is clear that the line with input 1 is tested for stuck at 0 whereas the input with value 0 will not be tested by this input pattern. We proceed in this way from the primary outputs ascertaining whether or not the line (device) is detected in its failure by this test, passing through the entire circuit to the primary input. This is thus a 1-pass-type of simulator. TESTDETECT for the acyclic case has been programmed in APL and run extensively. (Roth, Bouricius & Schneider, 1967)

TESTDETECT takes the order of n steps for a piece of logic having n devices. A simulator requires n^2 steps: for each failure one must determine the signal on each line of the circuit: n failures multiplied by n devices equals n^2.

Now we will outline the procedure for the case of the cyclic circuit and use the figure 2 as an illustration. Initially we assume that all lines have a signal which is unknown - let this condition by represented by the symbol x. In the case of cyclic logic the test will in general be a sequence. Assume that we apply the first input pattern to the primary inputs with all other lines being x. In general, some lines will be determined immediately by this primary input pattern. On the other hand there will be some for which this is not the case, for example, suppose that a given line is the output of an OR with two inputs, one having the value x and other, the value 0. Figure 2 shows a cyclic circuit (the registers required for DDF are omitted for simplicity). Figure 3 shows cyclic TESTDETECT operating for the first input pattern t1. Figure 4 shows it for the second t2.

The general procedure is as follows:

(1) The resulting signal 1 or 0, on each line if determined by t1 is computed; if not determined, it is assigned to value x. Clearly the input pattern can test no line assigned x.

(2) One then reasons backwards from the primary Output lines (here only line 10): if the test pattern causes a signal a, a = 0 or 1, on the PO line, then clearly this line is tested for the failure stuck-at-$\bar{a}$.

(3) Having determined failure detection for the PO lines, we next examine the lines ℓ feeding PO blocks (lines). If the PO block is an OR or NOR then ℓ is tested only if the signals on all other inputs to the block is 0; if an AND or a NAND, then all other lines feeding the block must be 1. If such a line is testable then it is tested for $\bar{a}$ if t induces a signal a on ℓ.

(4) This procedure is pushed level by level. If a line fans out to

more than one block then one must, as with acyclic TESTDETECT, proceed forward from the line to the point(s) of reconvergence. Nevertheless the process is very rapid.

(5) Having made the determination for the signals 1, 0 or x which the first input pattern causes, plus the ensemble of failures detected by t1, one similarly applies the second input pattern t2 to determine, together with the signals established on the feedback loops by t1, the signals induced on the lines, together with as above, the failure detected by t2-preceeded-by-t1.

(6) One continues similarly through the entire given sequence of test patterns, t1, t2,..., to determine all failures detected by this test.

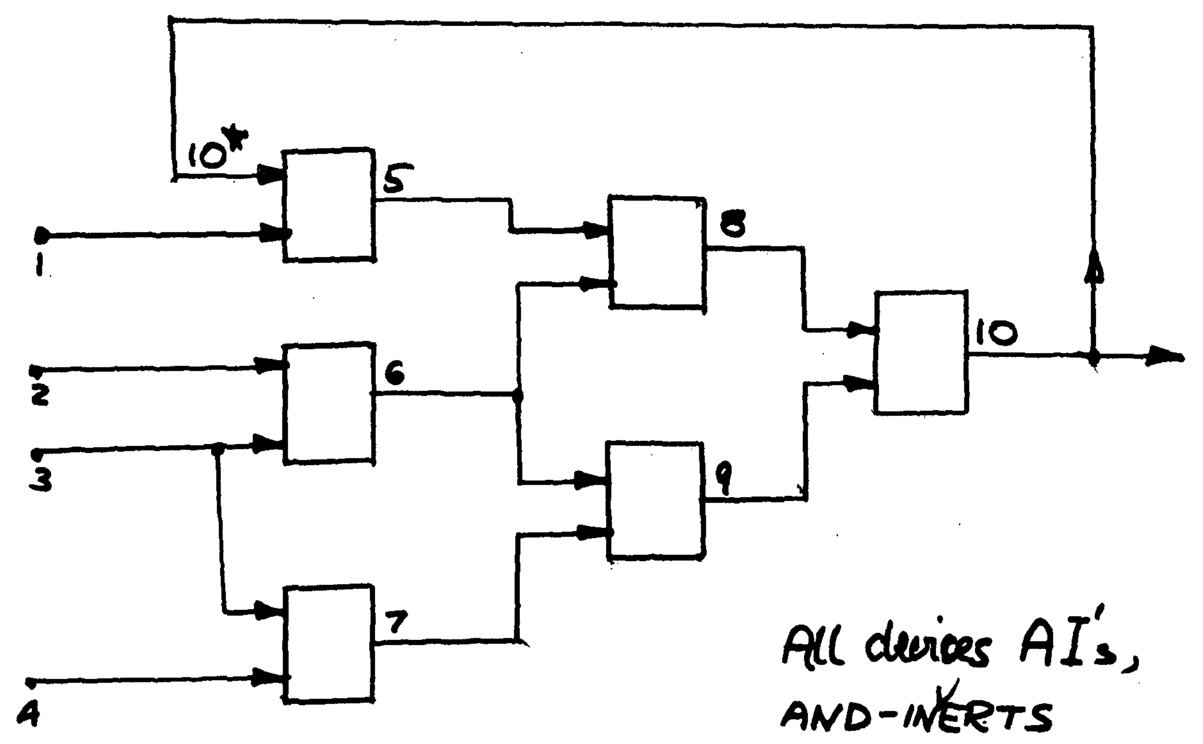
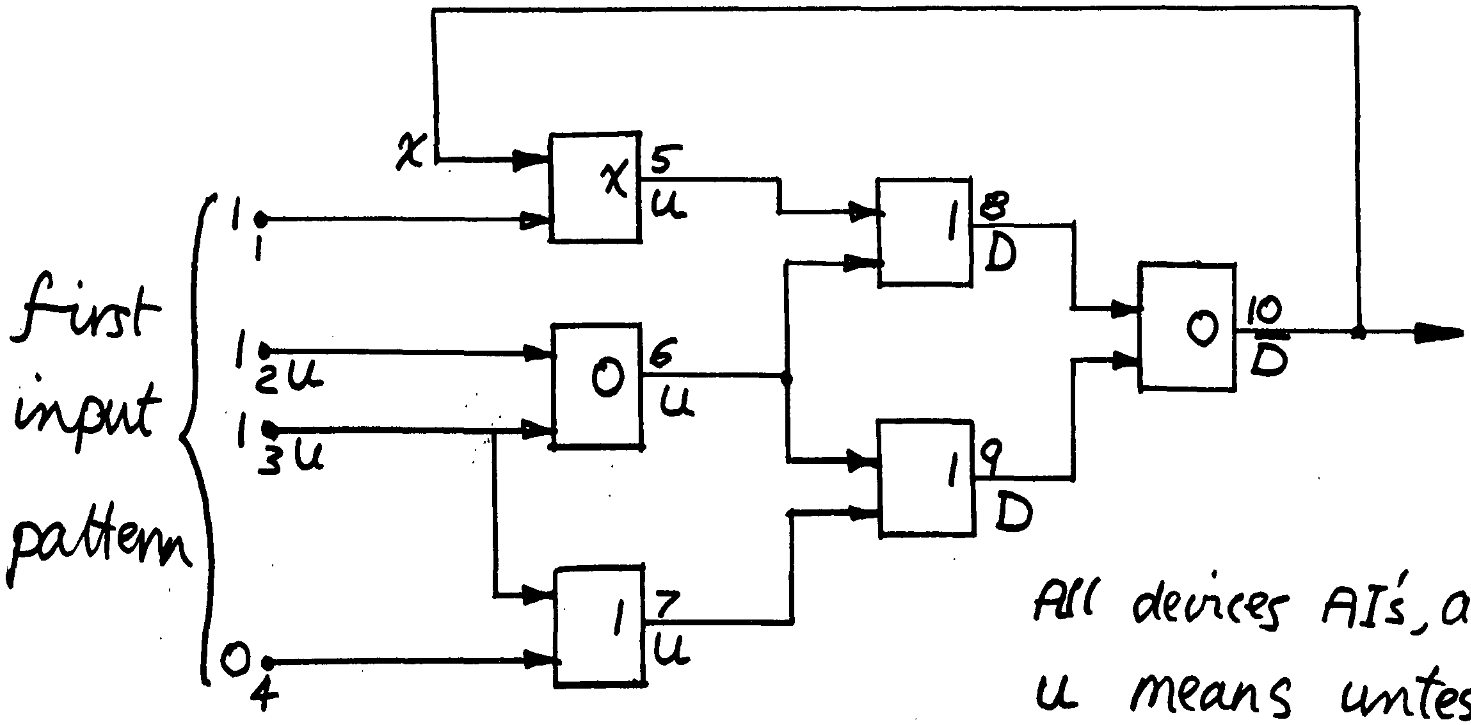


Fig 2 CYCLIC TESTDETECT — example



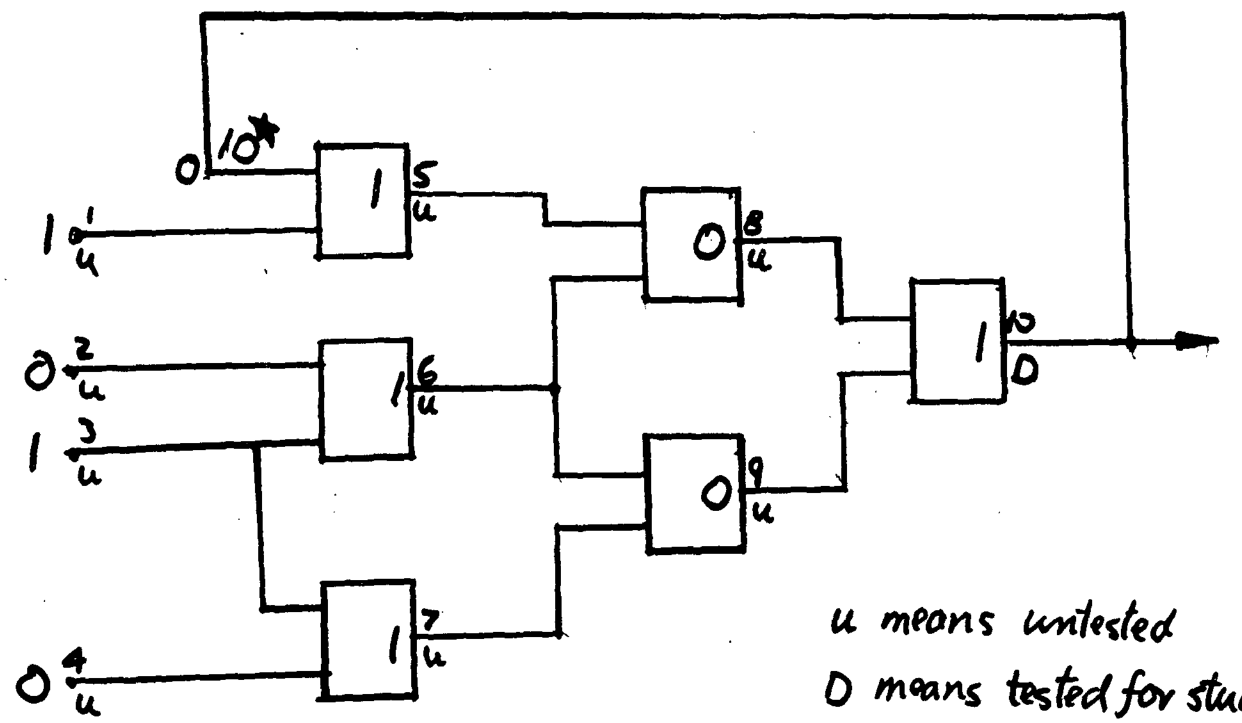
All devices AI's, and-inverts

u means untested

D means tested stuck-at-0

$\bar{D}$ means tested stuck-at-1

Fig 3 CYCLIC TESTDETECT – example.



u means untested

D means tested for stuck at 0

Fig 4 CYCLIC TESTDETECT Values assumed by second input pattern

Section 4. The Universal Function Schema

What follows is a report on the Universal Function Schema, previously unpublished.

A UNIVERSAL FUNCTIONAL OBJECT

by

J. Paul Roth

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

ABSTRACT: This paper presents a scheme for realizing any function, combinational or sequential, in a single universal function scheme, termed the universal function object UF. This universal functional scheme is addressed to the problem of the proliferation of the number of parts (cards, chips) necessary for conventional implementation in an LSI technology of a computer system (terminal, or peripheral systems). A UF implementation would use only one or a few, the various specializations being effected by insertion of the appropriate "connection vector". To use the universal functional scheme of implementation, the designer would proceed in conventional manner to produce a conventional design; a simple program would then faithfully translate this into a UF design. The UF implementation will use about ten times more circuits than a conventional implementation, regardless of the size of the design. However, there will be only one (or a few) distinct cards to form, so that actual manufacturing and stocking costs should be drastically reduced. Another feature of the UF approach is the following. Suppose that a particular circuit in the UF has failed, which fact we would have detected and diagnosed by precomputed and stored tests [4]. Then we have "general-purpose spares" and, by means of a new connection vector cv, logically disconnect the failing circuit, appropriately a connect spare properly biased by cv, thus substituting for the failing circuit. This procedure could be used both at manufacture, to increase the yields, as well as in the field, to achieve automatic repair. This work was supported in part by the Jet Propulsion Laboratory, California Institute of Technology, under the National Aeronautics and Space Administration Contract NAS-7-100.

In a recent disclosure I showed [1] how to realize any (finite) function, sequential or combinational, by means of a single type of primitive functional object, essentially by appropriately assigning values to strategically placed memory elements which constitute a part of this primitive, called a universal functional element UF.

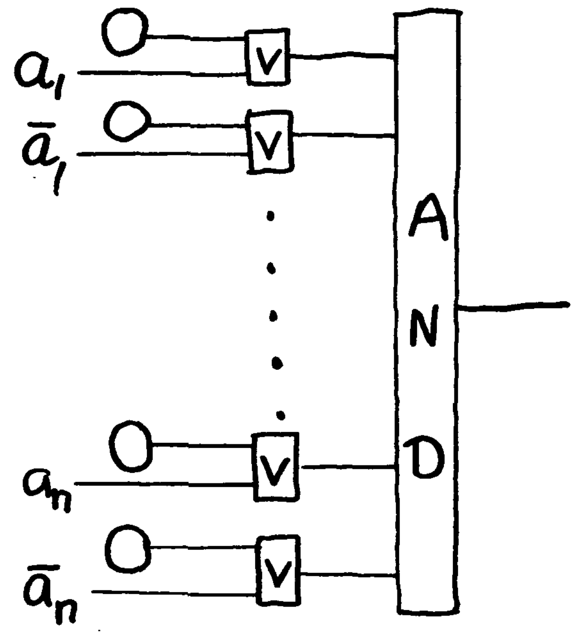
For combinational functions, this type of implementation utilized, however, only two levels of logic, at least in its most literal interpretation. For some functions, such as the parity-check function of any substantial number of variables, this might be a prohibitively costly method of implementation, although a way around this particular function, the parity-check, was given in [1].

This first extension given here removes the limitation to two levels to the UF; it allows, just as does ordinary "fixed-logic" implementation, many levels of logic in any given form of implementation. It extends directly to the sequential case, in which case loops (controlled) are introduced.

It appears at this stage in the development of these ideas that actual large-scale logical design using UF's will most expeditiously be done by having a variety of UF's, of differing size and complexity, in spite of the fact that any one sufficiently large UF type would suffice for any job - it is a matter of efficiency.

The second extension in this note addresses itself to the repairability of the universal functional object UF. We shall consider first the combinational case implemented in two levels the general scheme is shown in Figures 1 and 2.

It is also clear how Engineering Changes ED's, corrections or improvements to the design, can be effected merely by a change to the connection vector: no hardware scapping, the ED's simply effected and tested out.



Here the individual circles represent bits of memory, possibly arranged in shift-register style: if for a given input a_i, it is desired that a_i be ANDed in this universal AND, then the corresponding memory element feeding the OR-block, which a_i also feeds, is made equal to 0. If not, then the contacts of this memory element is made equal to 1, suppressing thereby the a_i. Similarly one treats the complementary variable $\bar{a}_i$. If it is desired that neither a_i nor $\bar{a}_i$, feed this AND then the memory bits of each is set equal to 1.

Figure 1. The Universal AND block UA

Let us take a few examples to see how we would design logic in this two-level or, as we prefer to call it, one-stage method.

a	b	A
1	0	1
0	1	1

exclusive-or

a	b	c	d	B	C	D
1	0	1	X	1	X	1
X	X	0	0	1	1	1
1	1	X	1	X	1	1
0	0	1	1	X	X	1

some other function

Figure 2. Tabular or cubical description of two functions.

In Fig. 2, the function on the left is the exclusive-or, with inputs a, b and output A.

That on the right is a 4-input 3-output function. The symbol x on the left means that with the other variables at the specified values, the output described on the right will be maintained <u>regardless</u> of the values assigned to the input variables having the value x. On the right of the vertical line, the output side, an x means that that term is not to feed that particular output [2]. IBM System 360 Program MOM exists to minimize the number of these rows (cubes) for any given (multiple-input, multiple-output) function [3].

Automatic-Repair on the Universal Function Object

Before we describe the case when the function is sequential (hence has memory) we describe a simple means of using the UF so as to be able in effect to remove or disconnect failing logic elements of the UF and to codify spare general-purpose elements on the UF and to connect it, again by purely logical means, so that it performs precisely the function formerly done by the failing element.

The exclusive-or is implemented for the universal element as shown in Figure 3.

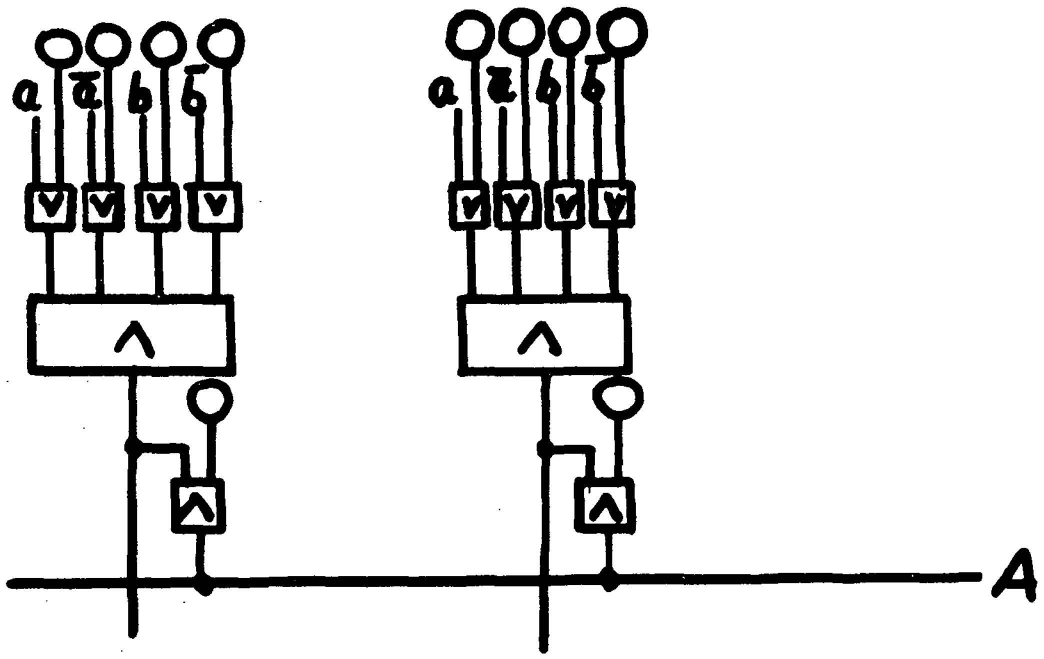


Figure 3. UF Implementation of exclusive-or.

Tests to detect and diagnose failures can be computed by the D-algorithm with particular ease because of the great control over each logical element.

Imagine then that we have n active Universal AND blocks and along with these general-purpose spares "floating", i.e. not contributing to the outputs; see Figure 5.

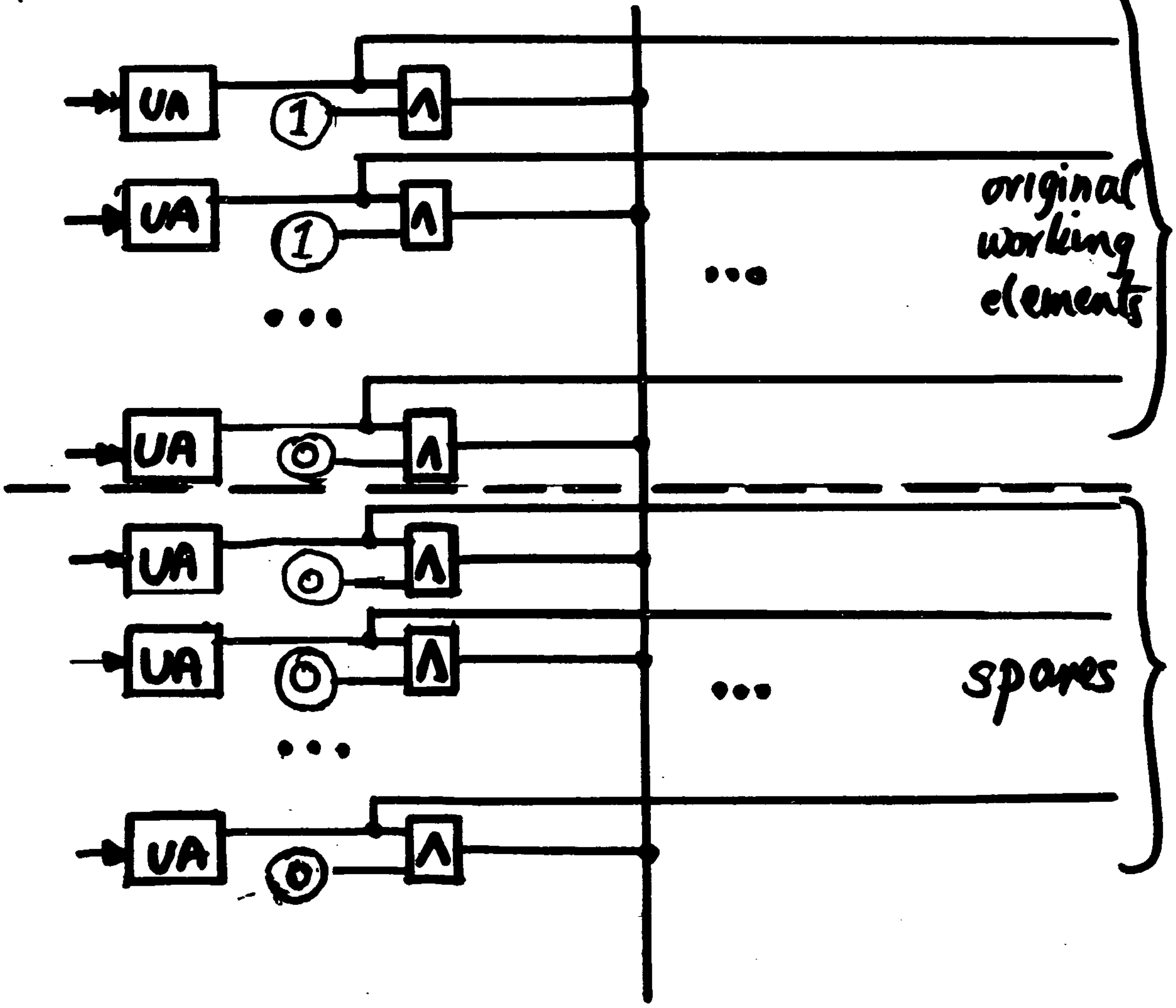


Figure 5. UF with general-purpose spares

Originally the r general-purpose spares are switched off, which condition can be achieved, for example, by setting the memory elements for the output contact equal to 0.

Suppose then that one of the original working elements is found to be malfunctioning, e.g. by application of a complete set of tests ("complete" means giving 100% coverage).

First, this element must be switched out which is done, as above, by setting all of its memory elements for its outputs to 0.

Second the connection vector which this universal and block had, i.e. the pattern of its memory bits controlling its input configuration, must be impressed into the similar memory bits for the spare selected for this replacement operation. Likewise the pattern of the memory output connections must similarly be duplicated.

Clearly this replacement scheme can be continued until all the spares are used up. It is worth noting that this procedure could be used in the manufacturing process, so that a given part need not be prefect - only a sufficient number of them. This would raise the yield.

Suppose that we call the output variables from the first stage (two levels of logic) B. Then if, as we shall discuss shortly, the B variables were used as inputs to subsequent stages of logic precisely the same replacement scheme could be to "repair" any faulty "B-logic". We show this by the following diagram and argument.

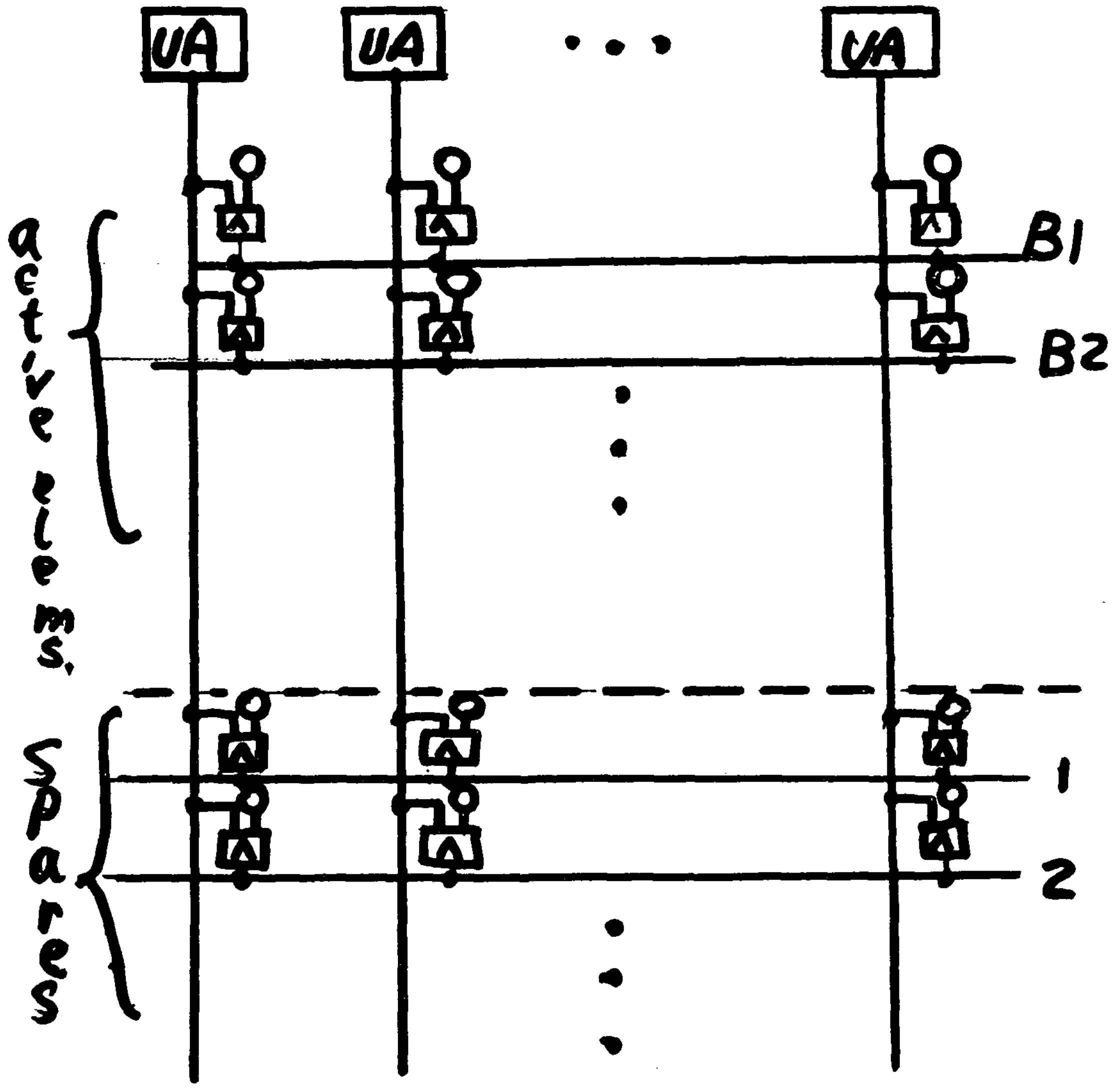


Figure 6. Illustration of operation of replacement of one of the outputs of the first stage.

We refer to the figure: If B1 should fail, then it would be logically disconnected and spare 1 would be invoked to serve as its substitute, with the outputs from the first stage made the same for the spare as for the line B1 which it is replacing.

Note: if in the second stage (and similarly for later stages) it were desired that $\overline{B1}$, the complement of B1, be an input to some vertex function, say an AND, this could easily be achieved by including a negation element with B1 as an input. It could also be achieved, however, by observing that, if the outputs to B1 are denoted as $b_1,\ldots,b_k$, then $B1 = b_1 \vee b_2 \vee \ldots \vee b_k$, so that $\overline{B1} = \quad (b_1,b_2,\ldots,b_k)$; thus if a NAND function is provided, $\overline{B1}$ can be easily provided, and independently from the formation of B1.

Note: a failure in the memory cell could equally well be considered as a failure in the appropriate logic block and treated, by replacement, in the same manner as any failure of the logic block.

Take now a function which is implemented in n levels

$$\begin{aligned} b &= B\ (a) \\ C &= B(b,a) \\ d &= D(c,b,a) \qquad (a,b,c,\ldots,z \text{ all binary vectors}) \\ &\vdots \\ z &= Z(y,\ldots,a) \end{aligned}$$

We would then have z stages in the Universal Functional Object UF appropriately biased; note that there may be several levels of logic within each stage. In the interest of simplicity rather than draw a general circuit for the above functional expression, we will give two detailed examples of UF - implementation, namely a four-bit parity check circuit and an adder of two two-bit numbers.

It is clear how these schemes must be extended to account for larger numbers of inputs for the same functions.

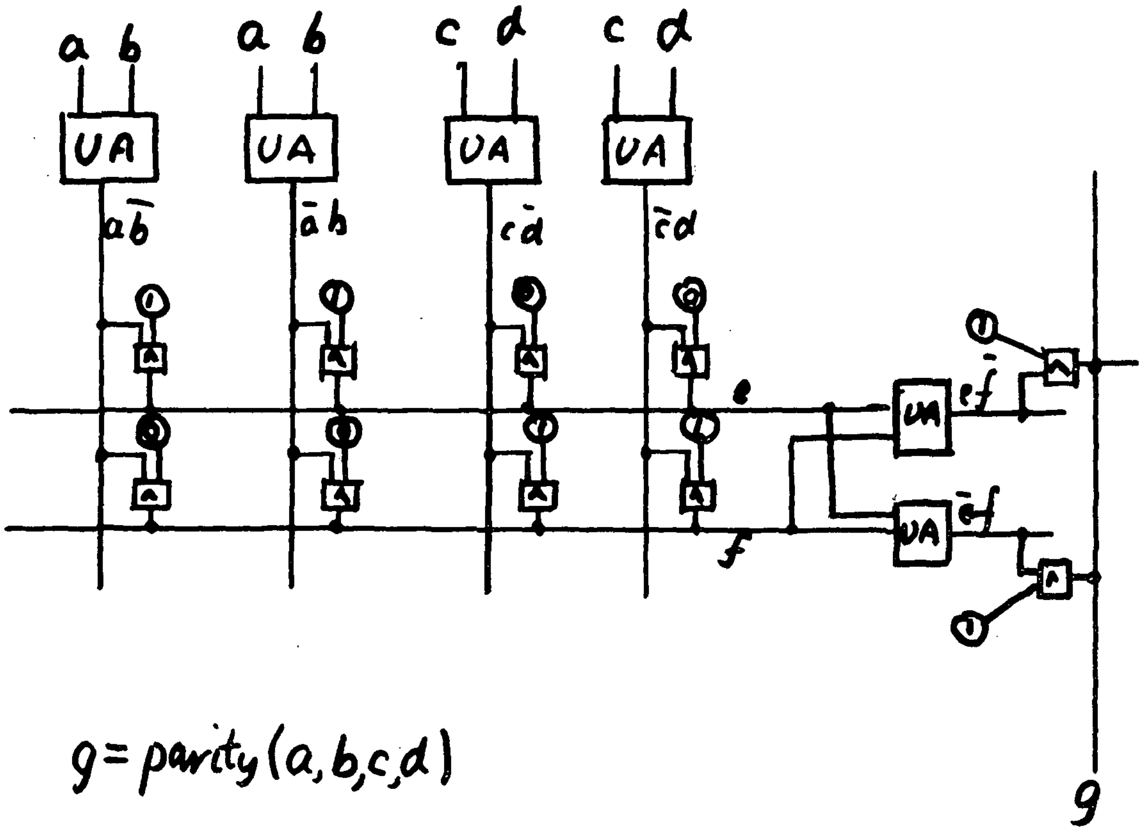


Figure 7. UF -Implementation of a 4-bit parity check circuit.

first bits		first sum	first carry
A	a	S_1	C_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

second bits		first carry	second sum	second carry
B	b	C_1	S_2	C_2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 8. Function table for a 2-bit adder.

Fig. 9 is a straight-forward implementation of a two-bit adder. It may be seen that this implementation uses three exclusive-or's and two AND's both of which we have given UF implementations.

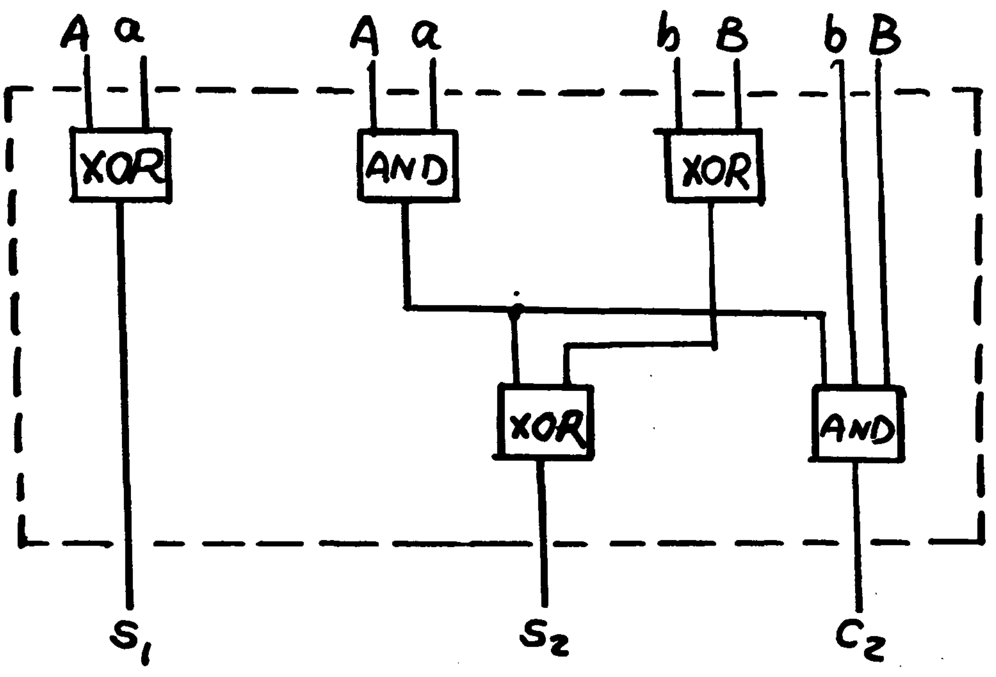


Figure 9. Standard implementation of a 2-bit adder.

We have seen how to implement each of these functions, AND and XOR are implemented on a UF ; for the sake of completeness, however, we shall make an explicit UF -implementation.

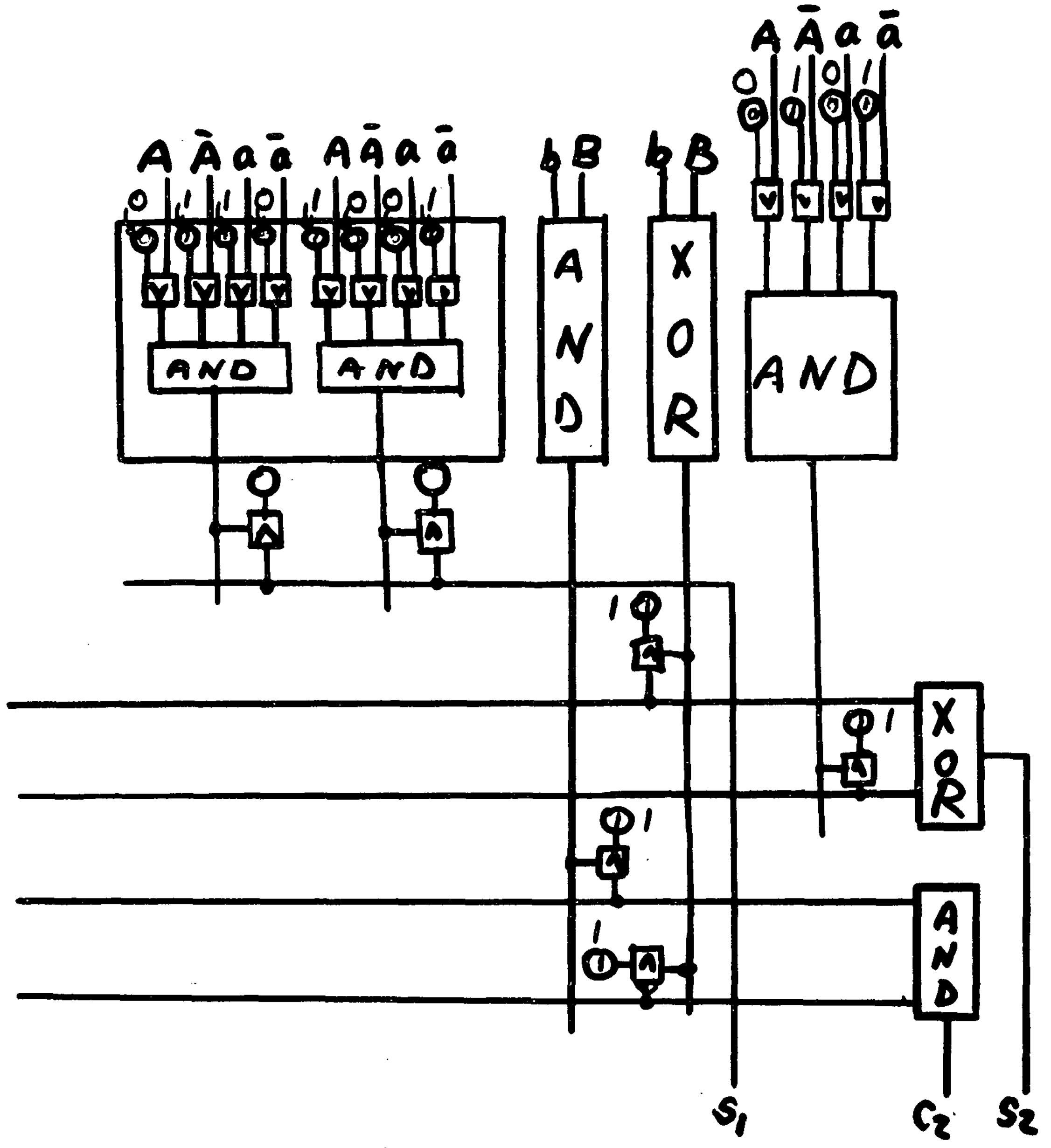


Figure 10. Translation of adder implementation of Fig. 9 into a UF -implementation.

In this picture not all of the memory cells and other lines are shown: only enough so as to exhibit the scheme clearly.

The scheme for having memory or feedback is very simple. Imagine a line y emanating from say the first stage of implementation and that it is desired, possibly upon appropriate delay or arrival of a timing pulse, to feed this signal back to the input of this stage. Then y is fed into an AND block together with a memory element, say m. If it is desired that y be fed back and hence its value "remembered" then m is set to 1; otherwise 0. Then the output of this AND - call it y* - is then fed into an OR whose other input is a primary input p. When y = 0 then clearly p is transmitted through the OR-block so that with respect to this line at least the circuit is behaving combinationally. When, however, m = 1 then p is set to 0 and y comes through the OR. This is the basic scheme as described in [1].

If input lines to the AND's in the first stage are at a premium, one might use the following selection scheme.

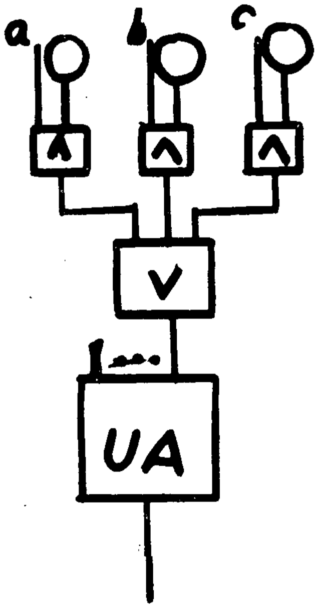


Figure 11.

Here we indicate a particular input line to the Universal AND block and here any of the inputs a,b,...,c may be selected, in general exactly one of them, by appropriate selection of its characteristic value: 1 0,...0 or 0 1 0, ...0,..., or 0,... 0 1.

This device can mitigate against the building up of an excessive number of inputs to the UA's.

In general each stage would have a suitable number of designated output lines which could be feedback lines. Selections could be made among these along the following lines,

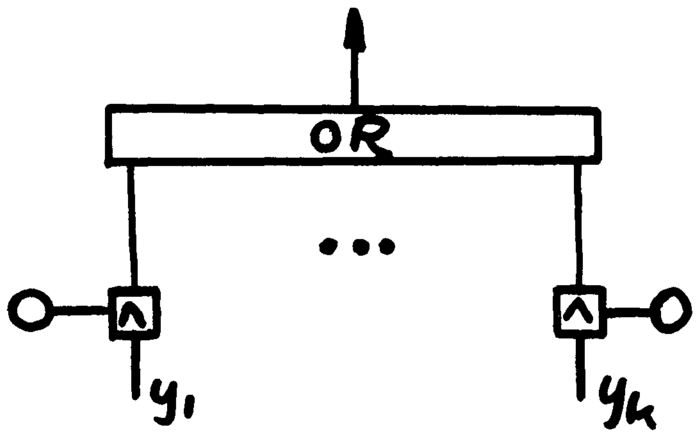


Fig. 12. Selection for feedback among the outputs of a given stage with the memory contents being 1 0...0 or 0 1 0...0... or 0...01.

It would probably be infeasible to have one or more output lines going from each stage i back to stage i - 1 and one could easily cascade back stage-by-stage although for fast applications this might be inacceptable.

REMARK: it is clear than an Engineering Change can be effected by changing the connection vector for the affected UF ; this will greatly reduce the cost effecting engineering changes.

Remark: It is quite straightforward to transform any simplex or ordinary logic design into a UF -design which is "isomorphic" to the simplex design in the sense that when the connection vector is appropriately assigned the various levels and gates of the original design can be seen in the UF -design and vice versa. In face one could write a program to effect such a transformation.

BIBLIOGRAPHY

[1] J. P. Roth, "The Universal Functional Element", IBM Research Disclosure # Y0 870392, August 12, 1971

[2] ________________, RC 2007, February 6, 1968.

[3] Roth, Wagner and Junker, "An algorithm and a System/360 program (MOM) for the minimization the cost of multiple-output functions implemented in two levels), To appear.

[4] Algorithms for the Mechanization of Design I. Diagnosis, RC 1294, Oct. 1964.

[5] H. Fleisher, A. Weinberger, V. Winkler, "The Writable Personalized Chip", Computer Design, June, 1970, 59-66.

Section 5. MIN360

One of the first problems encountered by designers of switching systems was that which is called today minimization of two-level realizations. In its original form it was concerned with the minimization of contacts in "strict series-parallel" form. Manual procedures were developed, by Karnaugh, Quine and McCluskey, the latter being adapted to digital computation, for small problems at least. In their original formulation the problem was for a single output function, that is to say, the minimization in normal form of a single output function. Concrete problems, however, usually have many outputs. A typical problem would be a two-level circuit transforming from one code to another. This inherently is a multiple output problem. A geometric method, designed for digital computation, was described in RC-11. This was *programmed in a series called MIN1 through MIN6 over the period* 1958 to 1961. It was adapted early using a scheme due to Muller for multiple outputs. It could not be guaranteed the multiple output form, however, to obtain a minimum solution. Various approximations were incorporated into the program, for example MAP, standing for Minimum APproximate. This program proved to be highly useful on problems out of the range of the strict minimization procedure. The MIN system of programs received extensive usage within IBM. (RC 1425).

In 1966, the Jet Propulsion Laboratory, in response to its needs for two-level realizations made a search, which the aid of IBM, for a decent minimization program. MIN 6 was tried and as a result of experiments modified so as to be more convenient for JPL usage. As a result of this renewed interest in minimization, we became interested again in the two-level multiple-output minimization problem, and using a different representation for cubes - singular cubes - we were able to extend the extraction algorithm to the multiple output case. (RC 2007, 2280) This algorithm was first programmed in APL and proved out very small problems. It was then reprogrammed by Leroy Junker in PL/1 with assembler language subroutines and run on the Model IBM System/360 Model 91.

In addition new approximate procedures and options were developed. For example, for Functional Memory the appropriate cost function is not the classical one, the number of input lines, plus the number of output lines, but rather the total number of cubes in the cover - this would correspond to the number of words in the Functional Memory realization of the function.

We will now describe how the program may be used in its various options; for simplicity we will call MIN360 by the abbreviated name MIN.

First let us describe the input format, which is the same as the output format. Suppose that we have a function of r binary inputs and s binary outputs. The cubical cover is a description of the input-output relations for this particular function. One mode of description, for example, would be to list all 2^r input patterns with their corresponding outputs if any. For r large this may not be practically possible. For functions of large numbers of variables, it is most often the case that relatively few combinations of the input will determine the output. Suppose for example that the first p of the inputs, if all 1, will cause each output to be 1. This would be represented as a string of p 1's followed a string of r-p x's followed by a vertical slash followed by s 1's. The minimization algorithm MIN360 is a function of the care conditions the don't-care conditions the mode, which will be explained, the cost function, a cost bound and truncation number. Thus we may write the solution s = MIN360 (c,d, mode, bound, truncation)

The care conditions c, don't-care conditions d, have already been described. The <u>mode</u> may be EXACT, meaning that a minimum will be obtained; PSEUDO meaning that a simplification of the algorithm will be used; in PSEUDO, after an extremal is computed it is thrown entirely into the developing solution s, whereas in the exact solution only that portion thereof which contains distinguished vertices may be thrown in. This procedure hastens the computation of a solution. After a solution is obtained, redundant outputs are removed. This is similar to the MIN6 mode for multiple outputs. Although it is in general faster than EXACT, the solution cannot be guaranteed to be a minimum. We ran a JPL problem for 2 hours in the EXACT and the PSEUDOmode; the EXACT mode obtained a solution of cost 409, whereas for the PSEUDOmode, the cost was about 427.

As indicated the cost function varies from one technology to another; the classical cost function counts the number of 1's and 0's in the input part of the cube in the solution and the number of 1's in the output. If one were to design a two-level AND-OR realization this would be the appropriate cost.

With the costbound option, cbound, if one assigns a costbound k to the developing solution, as soon as k is exceeded that part of the search is terminated. Under the truncation option, one assigns a number t and as soon as the branching level exceeds t, the computation ceases: one uses whatever solution or solutions of those computed up to that point.

The mode may take on the values EXACT, PSEUDO and APPROX. We will now explain APPROX. First a word, however, about the exact minimization procedures. For very large problems it is frequently the case that "branching" occurs. This means that a decision tree must be examined, each node of which involves, in general, considerable computation. Thus, as the size and complexity of the problem to be solved increases, the running time for the program correspondingly increases. Since we are interested in solving quite large problems - the data format for the program allows problems having up to 32 inputs and 32 outputs - it is important to have an APPROXimation which will get an answer hopefully close to a minimum, and in a reasonable, perferably very short, amount of time.

The APPROXimate procedure starts, as all MIN options, with a cover C of all input conditions for which an output should be 1: this is referred to as the ON-array; also either a cover of the DON'T-CARE's, i.e. input conditions together with those corresponding output conditions for which one does not care or is indifferent to the corresponding output or else a cover of the OFF-array i.e. a cover of input conditions or patterns for which some outputs are zero. For example, for 3 inputs and 2 outputs one might have

C, ON-array =

inputs	outputs
1 x 0	1 x
x 1 1	x 1
0 0 0	1 1

input	output
1 0 1	1 1

= D, don't -care array

1. The first step of the algorithm is to take the first cube of the ON-array, in our example 1 x 0 | 1 x and a coface of the first coordinate, i.e. change the first coordinate to an x if it is not already x. Then one "sharps" away the cubes of the cover; the #-product is a procedure for finding, for a and b cubes, a cover of the set of all vertices of a that are not contained in b. If C is a set of cubes then a # C is an iterated product of a with all cubes of C. See RC 2008 for details and an explanation.

If the #-product of the cofaced cube a of the initial cover and the initial cover C and D is empty, then changing this coordinate to an x is a valid step and a is replaced by the larger cube (in general covering more) and the procedure is repeated for each input coordinate of a which is not x. After this is completed, each output coordinate which is x is changed provisionally to a 1 and again the #-product of the cube modified, with the original covers C and D is used to ascertain whether or not this change is valid; this modification is done again for each output coordinate, one after the other. This cofacing operation is done for each cube of C: call the result E.

2. The <u>second step</u> ascertains if any cubes of E are redundant. This is done by performing for each cube e of E the test

e # ((E - e) v D.)

If the product is ϕ then e is redundant; e is then removed from E. This procedure is applied to every cube of E: this process produces a cover F which is irredundant, i.e. no cube can be deleted and still remain a correct expression of the original function.

3. The <u>third step</u> is to ascertain if any output coordinates of cubes of F are redundant. This is done by changing, in a given cube with respect to a given output coordinate which is 1 for this cube, all other output coordinates, which are 1, to an x and forming from that the #-product with all other cubes of the cover F plus the Don't-care

cover D. If this product is ϕ (empty) then this particular output coordinate is redundant and may be removed, i.e. changed to an x. The process is performed for every output coordinate whose value is 1 for every (singular) cube of F. The resulting cover, G, is an irredundant prime-cube cover and, experiment by computer shows, gives covers whose cost, being the number of cubes in G in many applications, is remarkably near to a minimum.

In fact for large problems, say in excess of 12 inputs and 12 outputs, we will inevitably use this APPROX made. The previous quarterly report gave running times for the computation on IBM System 360/91 of minimum covers, for approximate minimum covers, for a fairly wide range of problems, several of them being problems originating with M. Perlman at JPL. A manual for MIN360 is under preparation and will be sent to JPL upon its completion.

Section 6. Diagnosis of Failures in Mechanisms

Consider any component of a mechanism M. Assume that its "state" - position, velocity, pressure, tension, ... - can be characterized by a quantum number, assumed with no loss of generality to be in binary form, of bounded length. The state of a component c is a function of the "output states" of certain other components of M which directly affect it, plus possibly its own "previous" state. For simple c this function may be expressed as a function-table or function-ensemble of function-pairs. The component function-tables constitute a computable characterization of the function or behavior of M. Essentially M is treated as a physical, mathematical interconnection of its components. A "failure" of a component c is any change which modifies the function it performs; a failure of M is a failure of any of its components. Primary Inputs PI of M are those output states or variables of mechanisms outside of M which determine the functioning of M given its structure. Primary Outputs PO are those output states of M which are measureable and perceptible outside of M and are the result of M's functioning.

A test T for failure F of M is a (sequence of) Primary Input patterns such that the corresponding PO pattern differs depending upon whether or not F has occurred. Problem: given a failure in M, find a test to detect it. It does not appear difficult in principle to adapt the D-calculus (quantum calculus) to this model of the functioning behavior and failure diagnosis of mechanisms. The hard part is to provide this functional description of a mechanism whose design may be recorded at best by conventional drawings, further to be able to do this in a Design-Automation mode. Also there is the problem of characterizing and describing the behavior of components subject to a particular failure.

Nevertheless it seems an approach novel to mechanical diagnostics is being pursued actively.

Bibliography

June, 1957, J. P. Roth, "Combinational Topological Methods in the Synthesis of Switching Circuits", IBM Research Center, Poughkeepsie, New York, RC-11

January, 1959, Ann E. Randlev, "The Use of the 704 Program for Finding Minimum Two Level OR-AND Circuits", IBM Research Center, Yorktown Heights, New York, RC-90.

September, 1961, Ann C. Ewing, J. Paul Roth, Eric G. Wagner, "Algorithms for Logical Design", AIEE Communications and Electronics.

October, 1964, J. Paul Roth, "Algorithms for the Mechanization of Design I Diagnosis", IBM Research Center, Yorktown Heights, New York, RC-1294.

June, 1965, J. P. Roth, "Systematic Design of Automata", IBM Research Center, Yorktown Heights, New York, RC-1425.

October, 1968, J. P. Roth, IBM Research Center, Yorktown Heights, New York, and M. Perlman, Jet Propulsion Laboratory, Pasadena, California, "Space Applications of a Minimization Algorithm", RC-2235.

June, 1970, H. Fleisher, A. Weinberger, V. Winkler, "The Writable Personalized Chip", Computer Design.

January, 1972, J. Paul Roth, IBM Research Center, Yorktown Heights, New York, "Theory of Cubical Complexes with Applications to Diagnosis and Algorithmic Description", RC-3675. Quarterly Report.